CS 111
Operating System Interfaces

Jon Eyolfson

April 3, 2019

1.1.0

You Wouldn’t Write in Binary, Would You?

```
0x7F 0x45 0x4C 0x46 0x02 0x01 0x01 0x03 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x02 0x00 0x3E 0x00 0x01 0x00 0x00 0x00 0x78 0x00 0x01 0x00
0x00 0x00 0x00 0x00 0x40 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x40 0x00 0x38 0x00
0x01 0x00 0x40 0x00 0x00 0x00 0x00 0x00 0x01 0x00 0x00 0x00 0x05 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x01 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x01 0x00 0x00 0x00 0x00 0x00 0xB2 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0xB2 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x00 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x48 0xC7 0xC0 0x01 0x00 0x00
0x00 0x48 0xC7 0xC7 0x01 0x00 0x00 0x00 0x48 0xC7 0xC6 0xA6 0x00 0x01
0x00 0x48 0xC7 0xC2 0x0C 0x00 0x00 0x00 0x0F 0x05 0x48 0xC7 0xC0 0xE7
0x00 0x00 0x00 0x48 0xC7 0xC7 0x00 0x00 0x00 0x00 0x0F 0x05 0x48 0x65
0x6C 0x6C 0x6F 0x20 0x77 0x6F 0x72 0x6C 0x64 0x0A
```

The Previous Binary Prints “Hello world” and Exits

- You need to dump the binary into a file and make it executable
  - Only works on Unix based operating systems running x86-64

- How could this be possible in 178 bytes?

Executable and Linkable Format (ELF)

The binary format for all Unix based operating systems

Always starts with the 4 bytes: `0x7F 0x45 0x4C 0x46`

or with ASCII encoding: `0x7F 'E' 'L' 'F'`

Followed by a byte signifying 32 or 64 bit architectures

then a byte signifying little or big endian

ELF File Header

Use: `readelf <filename>`

Contains the following:

- Information about the machine (e.g. the ISA)
- The entry point of the program
- Any program headers (required for executables)
- Any section headers (required for libraries)

The header is 64 bytes, so we still have to account for 114 more.

`readelf -h` for minimal “Hello world”

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 03 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - GNU
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x10078
  Start of program headers:          64 (bytes into file)
  Start of section headers:          0 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         1
  Size of section headers:           64 (bytes)
  Number of section headers:         0
  Section header string table index: 0
```

ELF Program Header

Tells the operating system how to load the executable:

- Which type? Examples:
    - Load directly into memory
    - Use dynamic linking (libraries)
    - Interpret the program
- Permissions? Read / Write / Execute
- Which virtual address to put it?
    - Note that you'll rarely ever use physical addresses (more on that later)

For "Hello world" we load everything into memory. One program header is 56 bytes. 58 bytes left.

`readelf -l` for minimal "Hello world"

```
Elf file type is EXEC (Executable file)
Entry point 0x10078
There is 1 program header, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000010000 0x0000000000010000
                 0x00000000000000b2 0x00000000000000b2  R E    0x100
```

“Hello world” Needs 2 System Calls

Use: `strace <filename>`

This shows all the system calls our program makes:

```
execve("./hello_world", ["./hello_world"], 0x7ffd0489de40 /* 46 vars */) = 0
write(1, "Hello world\n", 12)            = 12
exit_group(0)                            = ?
+++ exited with 0 +++
```

System Call API for "Hello world"

`strace` shows the API of system calls

- An API tells you what a function needs
- Does not tell you where or how to layout variables

The `write` system call's API is:

- A file descriptor to write bytes to
- An address to contiguous sequence of bytes
- How many bytes to write from the sequence

The `exit_group` system call's API is:

- An exit code for the program (0-255)

System Call ABI for Linux x86-64

Enter the kernel with a `syscall` instruction, with the following variables in registers:

- `rax` – System call number
- `rdi` – 1st argument
- `rsi` – 2nd argument
- `rdx` – 3rd argument
- `r10` – 4th argument
- `r8` – 5th argument
- `r9` – 6th argument

What are the limitations of this?

Note: other registers are not used, whether or not they’re saved isn’t important for us

Instructions for “Hello world”, Using the Linux x86-64 ABI

Plug in the next 46 bytes into a disassembler, such as: `https://onlinedisassembler.com/`

Our disassembled instructions:

```
mov rax,0x1
mov rdi,0x1
mov rsi,0x100a6
mov rdx,0xc
syscall
mov rax,0xe7
mov rdi,0x0
syscall
```

Finishing Up “Hello world” Example

The remaining 12 bytes is the “Hello world” string itself, ASCII encoded:

`0x48 0x65 0x6C 0x6C 0x6F 0x20 0x77 0x6F 0x72 0x6C 0x64 0x0A`

- Low level tip for letters: bit 5 is 0 for upper case, and 1 for lower case (values differ by 32)

This accounts for every single byte of our 178 byte program, let’s see what C does…

Can you already spot a difference between strings in our example compared to C?

Source Code for “Hello world” in C

```
#include <stdio.h>

int main(int argc, char **argv)
{
  printf("Hello world\n");
  return 0;
}
```

Compile with: `gcc hello_world.c -o hello_world_c`

What are other notable differences between this and our “Hello world”?

All Applications Perform System Calls, and May Pass Through Multiple Layers

Applications e.g. `Chrome`, `Firefox`, `LibreOffice`, `Steam`, `VLC`, `Wireshark`

Toolkits e.g. `GTK+`, `Qt`

System daemons e.g. `udev`, `logind`

Display servers e.g. `X11`, `wayland`

C standard library (libc) e.g. `printf`

Ring 3 instructions

User space

Kernel space

Ring 0 instructions

System calls (335 total) e.g. `open`, `read`, `write`, `close`, `exit_group`, `ioctl`

Process scheduling

Inter-process communication (IPC)

Memory management

Virtual files

Network

System Calls for “Hello world” in C, Finding Standard Library

```
execve("./hello_world_c", ["./hello_world_c"], 0x7ffcb3444f60 /* 46 vars */) = 0
brk(NULL)                                = 0x5636ab9ea000
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=149337, ...}) = 0
mmap(NULL, 149337, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f4d43846000
close(3)                                 = 0
openat(AT_FDCWD, "/usr/lib/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0000C"..., 832) = 832
lseek(3, 792, SEEK_SET)                  = 792
read(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0\201\336\t\36\251c\324"..., 68) = 68
fstat(3, {st_mode=S_IFREG|0755, st_size=2136840, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
  = 0x7f4d43844000
lseek(3, 792, SEEK_SET)                  = 792
read(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0\201\336\t\36\251c\324"..., 68) = 68
lseek(3, 864, SEEK_SET)                  = 864
read(3, "\4\0\0\0\20\0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0", 32) = 32
```

System Calls for “Hello world” in C, Loading Standard Library

```
mmap(NULL, 1848896, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f4d43680000
mprotect(0x7f4d436a2000, 1671168, PROT_NONE) = 0
mmap(0x7f4d436a2000, 1355776, PROT_READ|PROT_EXEC,
  MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x22000) = 0x7f4d436a2000
mmap(0x7f4d437ed000, 311296, PROT_READ,
  MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x16d000) = 0x7f4d437ed000
mmap(0x7f4d4383a000, 24576, PROT_READ|PROT_WRITE,
  MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1b9000) = 0x7f4d4383a000
mmap(0x7f4d43840000, 13888, PROT_READ|PROT_WRITE,
  MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f4d43840000
close(3)                                = 0
arch_prctl(ARCH_SET_FS, 0x7f4d43845500) = 0
mprotect(0x7f4d4383a000, 16384, PROT_READ) = 0
mprotect(0x5636a9abd000, 4096, PROT_READ) = 0
mprotect(0x7f4d43894000, 4096, PROT_READ) = 0
munmap(0x7f4d43846000, 149337)          = 0
```

System Calls for “Hello world” in C, Setting Up Heap and Printing

```
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0x1), ...}) = 0
brk(NULL)                                = 0x5636ab9ea000
brk(0x5636aba0b000)                      = 0x5636aba0b000
write(1, "Hello world\n", 12)            = 12
exit_group(0)                            = ?
+++ exited with 0 +++
```

The C version of “Hello world” ends with the exact same system calls we need

C Calling Convention for x86-64

System calls use registers, while C is stack based:

- Arguments pushed on the stack from right-to-left order
- `rax`, `rcx`, `rdx` are caller saved
- Remaining registers are callee saved

What advantages does this give us? Disadvantages?

System Calls are Rare in C

Mostly you'll be using functions from the C standard library instead

Most system calls have corresponding function calls in C, but may:

- Set `errno`
- Buffer reads and writes (reduce the number of system calls)
- Simplify interfaces (function combines two system calls)
- Add new features

Note: system calls are much more expensive than C function calls (need to enter kernel space)

System Call vs C Example: `exit`

For an `exit` (or `exit_group`) system call: the program stops at that point

For `exit` in C there's a feature to register functions to call on program exit: `atexit`

```c
#include <stdio.h>
#include <stdlib.h>

void fini(void)
{
  puts("Do fini");
}

int main(int argc, char **argv)
{
  atexit(fini);
  puts("Do main");
  return 0;
}
```

produces:

```
Do main
Do fini
```

Abstraction Example: Memory

Operating systems provide virtual memory

For example if you have 16 GiB of RAM the operating system can use these physical addresses:
`0x000000000–0x3FFFFFFFF`

As a programmer it seems you have access to all the system's memory

The kernel maintains a table of processes mapping virtual addresses to physical addresses

- Implemented with a hardware translation lookaside buffer (TLB) usually
- Often mapped by pages (4096 bytes)

What Does Virtual Memory Give Us

Ability to run any number of applications, any number of instances

Recall: an executable has a single entry address

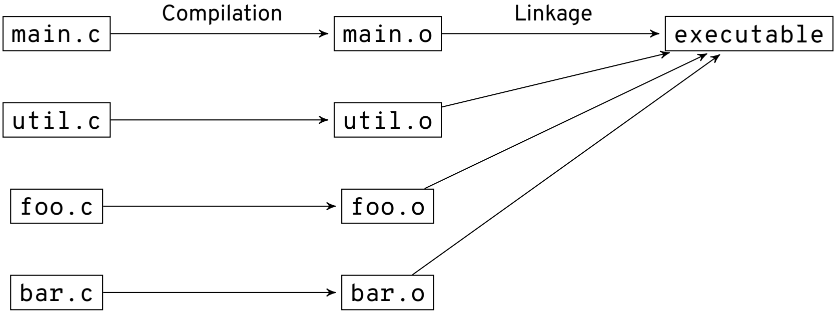
What's the alternative?

- Each application has to have exclusive access to a region of physical memory
- Any libraries cannot change size since addresses for libraries need to be fixed

Other benefits:

- Operating system can map virtual addresses to hardware other than physical memory

Normal Compilation in C



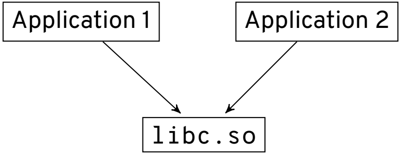
Note: object files (`.o`) are just ELF files with code for functions

Dynamic Libraries Are For Reusable Code

The C standard library is a dynamic library (`.so`), like any other on the system

- Basically a collection of `.o` files containing function definitions

Multiple applications can use the same library:



The operating system only loads `libc.so` in memory once, and shares it

- The same physical page corresponds to different virtual pages in processes

Useful Command Line Utilities for Dynamic Libraries

`ldd <executable>`

shows which dynamic libraries an executable uses

`objdump -T <library>`

shows the symbols (often just function names) that are in the library

You can also use `objdump -d` to disassemble the library

Static vs Dynamic Libraries

Another option is to statically link your code

Basically copies the `.o` files directly into the executable

The drawbacks compared to dynamic libraries:

- Statically linking prevents re-using libraries, commonly used libraries have many duplicates
- Any updates to a static library requires the executable to be recompiled

What are issues with dynamic libraries?

Dynamic Libraries Updates Can Break Executables with ABI Changes

An update to a dynamic library can easily cause an executable using it to crash

Consider the following in a dynamic library:

- A `struct` with multiple public fields corresponding to a specific data layout (C ABI)

An executable accesses the fields of the `struct` in the dynamic library

Now if the dynamic libraries reorders the fields

- The executable uses the old offsets and is now wrong

Note: this is OK if the dynamic library never exposes the fields of a `struct`

C Uses a Consistent ABI for `structs`

`structs` are laid out in memory with the fields matching the declaration order

- C compilers ensure the ABI of `structs` are the consistent for an architecture

Consider the following structures:

Library v1:

```
struct point {
  int y;
  int x;
};
```

Library v2:

```
struct point {
  int x;
  int y;
};
```

For Library v1 the x field is offset by 4 bytes from the start of `struct point`'s base

- For Library v2 it is offset by 0 bytes, and this difference will cause problems

Our Code Should Always Print 3, then 9

```c
#include <stdio.h>
#include <stdlib.h>

#include "libpoint.h"

int main(int argc, char **argv)
{
  struct point *p = malloc(sizeof(struct point));
  p->x = 3;
  p->y = 4;
  printf("p.x = %d\n", p->x);
  squareX(p);
  printf("p.x = %d\n", p->x);
  return 0;
}
```

Mismatached Versions of This Library Causes Unexpected Results

The definition of `sqaureX` in both libraries is:

```
void squareX(struct point *p) {
  p->x *= p->x;
}
```

In v1 of the library, this code squares the `int` at offset 4

  v2 squares the `int` at offset 0

If we compile our code against v1, all our `x` accesses are to offset 4 of the `struct`

  Compiling against v2 changes all our `x` accesses to offset 0

Compiling against a version of the library and using another causes unexpected results:

  It will always print 3 followed by 3

    Matching the versions prints 3 followed 9, as expected

Try the Previous Example

It uses the `LD_LIBRARY_PATH` trick (mentioned again later) to simulate a library update

Run the following commands to see for yourself:

```
git clone https://github.com/eyolfson/talks
cd talks/2019/ucla-cs-111-lecture-2
make
make run_my_code_compiled_with_point_v1_with_point_v1
make run_my_code_compiled_with_point_v2_with_point_v1
make run_my_code_compiled_with_point_v1_with_point_v2
make run_my_code_compiled_with_point_v2_with_point_v2
```

Semantic Versioning Meets Developer's Expectations

From `https://semver.org/`, given a version number MAJOR.MINOR.PATCH, increment the:

- MAJOR version when you make incompatible API/ABI changes
- MINOR version when you add functionality in a backwards-compatible manner
- PATCH version when you make backwards-compatible bug fixes

Dynamic Libraries Allow Easier Debugging

Control dynamic linking with environment variables (e.g. `LD_LIBRARY_PATH` and `LD_PRELOAD`)

Consider the following example:

```c
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char **argv)
{
  int *x = malloc(sizeof(int));
  printf("x = %p\n", x);
  free(x);
  return 0;
}
```

We Can Monitor All Allocations with Our Own Library

Normal runs of `alloc_test` outputs:

```
x = 0x561116384260
```

Create `myalloc.so` that outputs all `malloc` and `free` calls

Now we run with `LD_PRELOAD=./myalloc.so ./alloc_test` which outputs:

```
Call to malloc(4) = 0x55c12aa40260
Call to malloc(1024) = 0x55c12aa40280
x = 0x55c12aa40260
Call to free(0x55c12aa40260)
```

Interesting, we did not make 2 `malloc` calls

Detecting Memory Leaks

`valgrind` is another useful tool to detect memory leaks from `malloc` and `free`

- Usage: `valgrind <executable>`

Here's a note from the `man` pages regarding what we saw:

The GNU C library (`libc.so`), which is used by all programs, may allocate memory for its own uses. Usually it doesn't bother to free that memory when the program ends—there would be no point, since the Linux kernel reclaims all process resources when a process exits anyway, so it would just slow things down.

Note: this does not excuse you from not calling `free`!

Abstraction Example: File Descriptors

File descriptors are just a number and may used to represent:

- Regular files
- Directories
- Block devices
- Character devices
- Sockets
- Named pipes

All of these can be used with `read` and `write` system calls

Kernel maintains a per-process file descriptor table

Standard File Descriptors for Unix

All command line executables use the following standard for file descriptors:

- `0` – `stdin` (Standard input)
- `1` – `stdout` (Standard output)
- `2` – `stderr` (Standard error)

The terminal emulators job is to:

- Translate key presses to bytes and write to `stdin`
- Display bytes read from `stdout` and `stderr`
- May redirect file descriptors between processes

Checking Open File Descriptors on Linux

`/proc/<PID>/fd` is a directory containing all open file descriptors for a process

`ps x` command shows a list of processes matching your user (lots of other flags)

A terminal emulator may give the output:

```
> ls -l /proc/21151/fd
0 -> /dev/tty1
1 -> /dev/tty1
2 -> /dev/tty1
```

`lsof <file>` shows you what processes have the file open

For example, processes using C: `lsof /lib/libc.so.6`

Lessons Learned Today

- Basic executable format (ELF files)
- Difference between API and ABI
- How to find all system calls
- Virtual memory and why it's needed
- Dynamic libraries and a comparison to static libraries
- File descriptors and standard conventions